# Deriving Safety Cases for the Formal Safety Certification of Automatically Generated Code

Nurlida Basir [1]

*ECS, Southampton University*
*Southampton, SO17 1BJ, UK*

Ewen Denney [2]

*USRA/RIACS, NASA Ames Research Center*
*Mountain View, CA 94035, USA*

Bernd Fischer [3]

*ECS, Southampton University*
*Southampton, SO17 1BJ, UK*

**Abstract**

We present an approach to systematically derive safety cases for automatically generated code from information collected during a formal, Hoare-style safety certification of the code. We use a generic safety case that is instantiated with respect to the certified safety property and the program. It is complemented by a static system safety case that argues the safety of the framework itself, in particular the correctness of the safety policy (i.e., Hoare rules) with respect to the safety property (i.e., safety claims) and the integrity of the certification system and its individual components. However, the safety case only makes explicit the formal and informal reasoning principles, and reveals the top-level assumptions and external dependencies that must be taken into account; the evidence still comes from the formal safety proofs.

*Keywords:* Automated code generation, Hoare logic, formal code certification, safety case, Goal Structuring Notation.

## 1 Introduction

Model-based design and automated code generation have become popular, but substantial obstacles remain to their more widespread adoption in safety-critical domains: since code generators are typically not qualified, there is no guarantee that their output is safe, and consequently the generated code still needs to be fully tested and certified. In principle, formal methods such as formal software safety certification [2] could then be used to

[1] Email: nb206r@ecs.soton.ac.uk . Supported by the IPTA Academic Training Scheme of the Ministry of Higher Education of the Malaysian Government.

[2] Email: Ewen.W.Denney@nasa.gov . Supported by NASA under awards NCC2-1426 and NNA07BB97C.

[3] Email: b.fischer@ecs.soton.ac.uk

demonstrate the required safety and integrity level. These rely on formal proofs as explicit evidence or *certificates* for the assurance claims, but typically require the developers to provide detailed logical annotations to construct the safety proofs. In previous work, we have addressed this problem and developed a technique to automatically infer these annotations for automatically generated code [4,5]. However, several problems remain. For automatically generated code it is particularly difficult to relate the proofs to the code; moreover, the proofs are the final stage of a complex process and typically contain many details. This complicates an intuitive understanding of the assurance claims provided by the proofs. The complexity of the involved tools can lead to unforeseen interactions (e.g., due to errors in syntax translations) and thus causes additional concerns about the trustworthiness of the assurance claims.

Here, we build on our previous work to address these problems. We present an approach currently under development to systematically (and ultimately automatically) derive safety cases from information collected during annotation inference. The approach is based on a generic, multi-tiered safety case that is instantiated with respect to a given safety property and program. The upper tier instantiates the framework for the given safety property. It refers to another, completely static safety case that argues the safety of the formal certification framework itself, in particular the correctness of the safety policy (i.e., Hoare rules) with respect to the safety property (i.e., safety claims). The lower tiers argue the safety of the program as governed by the safety property. However, they cannot be simply instantiated from a generic "blueprint" but need to be constructed individually to reflect the program structure. Fortunately, this can be done systematically because the overall argument structure is determined by the certification framework and directly follows the course the annotation inference algorithm takes through the program. The lower tiers refer to the safety proofs and to a system-wide safety case arguing the integrity of the certification system and its individual components such as the domain theory or the theorem prover.

This paper describes work still in progress. So far we have developed the overall structure of the generic program safety case and manually instantiated it for the first examples, using only information logged by the annotation inference algorithm. We expect that this process can be automated easily and that it will furthermore be straightforward to integrate with existing tools to construct safety cases such as Adelard's ASCE tool [1]. We have already started to develop the safety case for the certification system and its individual components, but not for the one for the framework. We believe that the combined safety cases (i.e., for the program, the formal framework, the certification system and its individual components, and the safety proofs) will clearly communicate the safety claims, key safety requirements, and evidence required to trust the generated code. We expect that this will alleviate any distrust in code generators, which is still a problem in using automated code generation techniques in safety-critical applications.

## 2 Formal Software Safety Certification

The purpose of formal software safety certification is to formally demonstrate that a program does not violate certain conditions during its execution. A *safety property* is an exact characterization of these conditions, based on the operational semantics of the programming language. A *safety policy* is a set of Hoare rules designed to show that safe programs satisfy the safety property of interest. The rules are formalized using the usual Hoare

triples extended with a "shadow" environment which records safety information related to the corresponding program variables, and a *safety predicate* that is added to the computed verification conditions (VCs) [2]. As a variant of program verification, formal software safety certification follows the same technical approach. A VC generator (VCG) traverses the code backwards and applies the Hoare rules to produce VCs, starting with the postcondition *true*. If all VCs are proven by an automated theorem prover (ATP), the program is safe wrt. the safety property.

Our example uses initialization safety but our framework can handle a variety other safety properties including absence of out-of-bounds array accesses and nil-pointer dereferences [2,6]. However, it is not restricted to showing exception freedom but can also encode domain-specific properties such as matrix symmetry or coordinate frame consistency, whose violation will not (immediately) cause a run-time exception but still renders the code unsafe.

## 3 Annotation Inference

Since the ATP has no access to the program internals, annotations must formalize all pertinent information that is required to prove that potentially unsafe use locations are in fact safe. If the program is safe, this information will be established at some definition location and maintained along all control-flow paths to the use locations. The aim of annotation inference is to "get information from definitions to uses", i.e., to find the endpoints of all such *def-use*-chains, to construct the formulae used in the annotations, and to annotate the program such that the VCG has the necessary information as it works its way back through the program. The notions of definitions and uses are specific to the given safety property. For initialization safety, definitions correspond to the different variable initializations while uses are statements which read a variable. For array bounds safety, definitions are the array declarations since the shadow variables get their values from the declared bounds, while uses are statements which access an array variable. We can exploit the idiomatic structure of automatically generated code (i.e., the fact that the code exhibits some regular structure beyond the syntax of the programming language and uses similar constructions for similar tasks) and use patterns to describe definitions, uses, and irrelevant code fragments.

The annotation inference algorithm first scans the code for relevant use locations. For each used variable, it builds an abstracted control flow graph where code fragments matching the patterns are collapsed into single nodes. It then follows all paths backwards from the variable's use nodes until it encounters either a cycle or a definition node for the variable. Paths that do not end in a definition are discarded and the remaining paths are traversed node by node. Annotations are added to all intermediate nodes that otherwise constitute barriers to the information flow before the definitions themselves are annotated. The form of the annotations is fully determined by the safety property and the known syntactic structure of the definitions, as described by the pattern.

## 4 Deriving Safety Cases via Annotation Inference

In our work, we consider each violation of the given safety property as a hazard. To demonstrate that this hazard can not lead to a system failure, we construct a safety case that argues that the safety property is in fact not violated and thus that the risk associated with this haz-

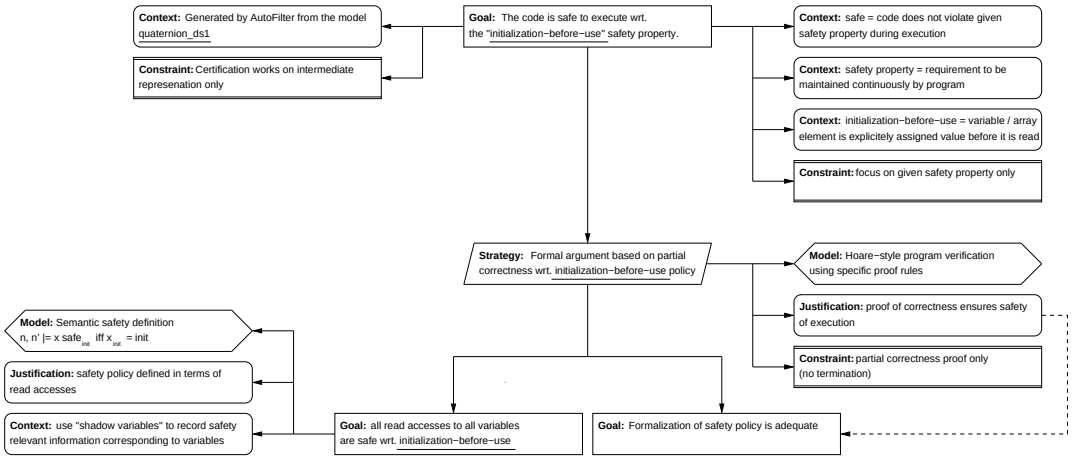


Fig. 1. Tier I of Derived Safety Case: Arguing the Approach

ard is controlled or mitigated. The high-level structure of this argument can be constructed from information collected by the annotation inference algorithm. However, the evidence still comes from the formal safety proofs. The safety case only makes explicit the formal and informal reasoning principles, and reveals the top-level assumptions and external dependencies that must be taken into account. It can thus be thought of as "structured reading guide" for the safety proofs.

Here, we provide a simplified overview of this safety case. We concentrate on its generic structure and describe its different tiers. We further concentrate on the program itself, leaving the remaining elements (i.e., for the formal framework, the certification system and its individual components, and the safety proofs) of the system-wide safety case for future work.

### *4.1 Tier I: Arguing the Approach*

Figure 1 shows the goal structure for the top tier of the safety case. It starts with the top-level safety goal (i.e., the safety of the generated code with respect to the safety property of interest) and shows how this is achieved by a formal argument based on the partial correctness of the generated code. The argument stresses the meaning of the Hoare-style framework. It uses contexts to explain the informal interpretation of key notions like "safe" and "safety property" and constraints to outline limitations of the approach, in particular the fact the certification works on an intermediate representation of the source code and only shows a single property. Hyperlinks refer to additional evidence in the form of documents containing, for example, the model from which the source code has been generated.

The key strategy at this tier and its model (i.e., a Hoare-style partial correctness proof using the dedicated proof rules of the init-before-use safety policy) as well as its limitations (i.e., no termination proof) are made explicit. The strategy reduces showing the safety of the whole program to showing the safety of all read accesses, which emerges as first subgoal. This is justified by the fact that the safety property is defined in terms of variable read accesses. The subgoal is further elaborated by a model of the semantic safety definition, which exactly defines what is meant by "safe", using the notion of shadow variables given as context. The strategy's second subgoal is to show that the safety policy adequately represents the safety property, which is also the foundation of the strategy's original justification

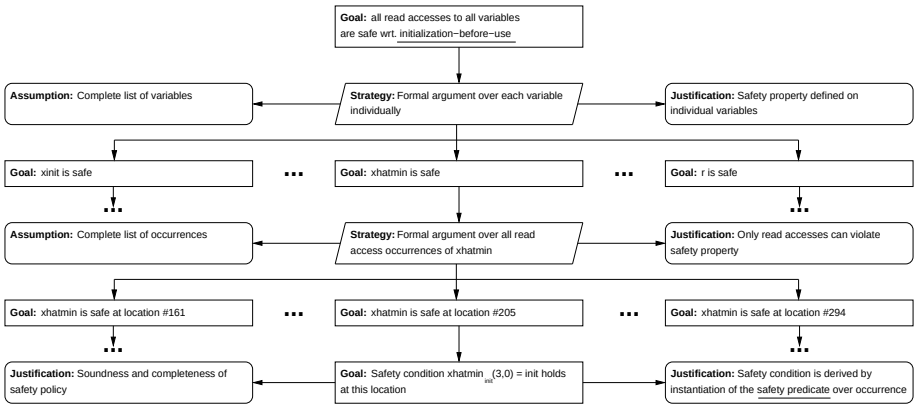


Fig. 2. Tier II of Derived Safety Case: Arguing over the Variables

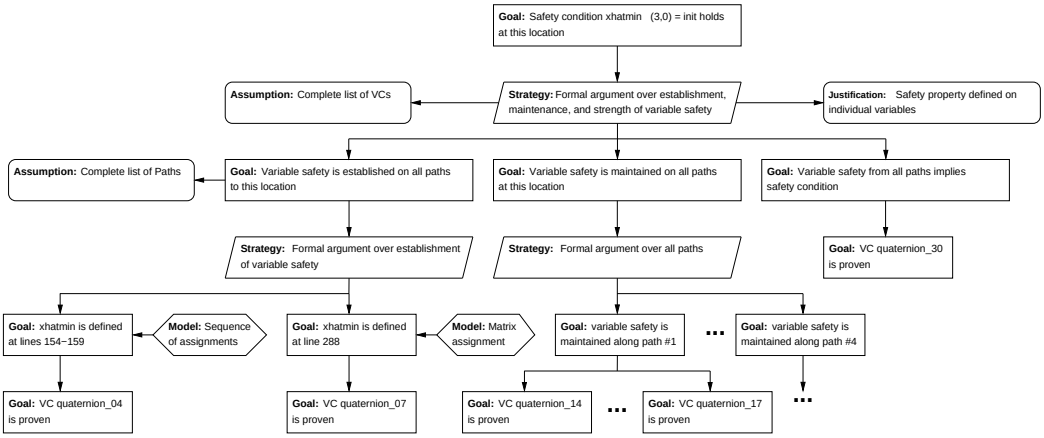


Fig. 3. Tier III of Derived Safety Case: Arguing over the Paths

(i.e., the claim that the proofs ensure the safe execution of the program). This subgoal is not elaborated further in this safety case but leads to the static framework safety case.

### *4.2 Tier II: Arguing over the Variables*

The second tier reduces the safety of all variables in two steps, first to the safety of each individual variable (justified by the fact that the safety property is defined on individual variables) and then to the safety of the individual occurrences. Note that the number of subgoals of both strategies (see Figure 2 for the goal structure) and the safety conditions are program-specific. This information is provided by the annotation inference.

Both strategies are predicated on the assumption that they iterate over the complete list of variables (resp. occurrences). Each individual occurrence then leads to a subgoal to show that the computed safety condition is valid at the location of the variable's occurrence. This reduction to a formal proof obligation is justified by the soundness and completeness of the safety policy; in addition, the specific form of the safety condition is also justified.

### *4.3 Tier III: Arguing over the Paths*

The final tier (see Figure 3 for the goal structure) argues the safety of each individual variable access, using a strategy based on establishing and maintaining appropriate invariants.

This directly reflects the course the annotation inference has taken through the code. The first subgoal is thus to show that the variable safety is established on all paths leading to the current location, using a formal argument over all definition locations. Here, the model for the subgoal corresponds to the pattern that was applied during annotation inference to match the definition. Each definition thus leads to a corresponding subgoal and then further to any number of VCs, although here only a single VC emerges in both cases.

The association of the VCs to the definition is based on tracing information added by the VCG. The second subgoal of the top-level strategy is to show that the established variable safety is maintained along all paths. This proceeds accordingly. The final subgoal is then to show that the variable safety implies the validity of the safety condition. This can again lead to any number of VCs. If (and only if) all VCs can be shown to hold, then the safety property holds for the entire program. The evidence for the VCs is provided by the formal proofs; we plan to convert these into safety cases as well.

## 5 Conclusions

Software development standards for safety-critical domains such as DO-178B [7] are typically process-oriented and require that code generators are qualified for application, often using an elaborate testing regime [8]. This time-consuming and expensive process slows down generator development and application. We believe that product-oriented assurance approaches are a viable alternative. Here, assurance is not implied by the trust into the generator but follows from an explicitly constructed argument for the generated code. We further believe that formal methods such as formal software safety certification can provide the highest level of assurance of the code's safety and integrity. However, the proofs by themselves are no panacea, and it is important to make explicit which claims are actually proven, and on which assumptions and reasoning principles both the claim and the proof rest. We believe that purely technical solutions such as proof checking [10] fall short of the assurance provide by our safety case. In fact, we consider the safety case only as a first step towards a full-fledged software certificate management system [3].

We have described work still in progress. So far, we have developed the overall structure of the generic program safety case and instantiated it manually. The example shown here uses code generated by our AutoFilter system [9], but the underlying annotation inference algorithm has also been applied to code generated from Matlab models using Real-Time Workshop, and we expect that the same derivation can be applied there as well. Current work involves constructing a system-wide safety case that covers all remaining components (i.e., for the formal framework, the certification system and its individual components, and the safety proofs) used in certification.

## References

[1] *ASCE home page* (2007), www.adelard.com/web/hnav/ASCE .

[2] Denney, E. and B. Fischer, *Correctness of source-level safety policies*, in: K. Araki, S. Gnesi and D. Mandrioli, editors, *Proc. FM 2003: Formal Methods*, LNCS **2805** (2003), pp. 894–913.

[3] Denney, E. and B. Fischer, *Software certification and software certificate management systems (position paper)*, in: *Proceedings of the ASE Workshop on Software Certificate Management Systems (SoftCeMent' 05)*, 2005, pp. 1–5.

[4] Denney, E. and B. Fischer, *Annotation inference for safety certification of automatically generated code (extended abstract)*, in: S. Uchitel and S. Easterbrook, editors, *Proc. 21st ASE* (2006), pp. 265–268.

[5] Denney, E. and B. Fischer, *A generic annotation inference algorithm for the safety certification of automatically generated code*, in: S. Jarzabek, D. C. Schmidt and T. L. Veldhuizen, editors, *Proc. Conf. Generative Programming and Component Engineering* (2006), pp. 121–130.

[6] Necula, G. C., *Proof-carrying code*, in: *Proc. 24th POPL* (1997), pp. 106–19.

[7] RTCA, "Software Considerations in Airborne Systems and Equipment Certification," RTCA, 1992.

[8] Stürmer, I. and M. Conrad, *Test suite design for code generation tools*, in: *Proceedings of 18th IEEE International Conference on Automated Software Engineering* (2003), pp. 286–290.

[9] Whittle, J. and J. Schumann, *Automating the implementation of Kalman filter algorithms*, ACM Transactions on Mathematical Software **30** (2004), pp. 434–453.

[10] Wong, W., *Validation of HOL proofs by proof checking*, Formal Methods in System Design: An International Journal **14** (1999), pp. 193–212.